

Optimization and Tuning on RS6000 SP

- (1) Think about the algorithm
- (2) Profile your application - xprofiler, gprof, prof, tprof
- (3) Be aware of hardware limits
- (4) Know your compiler
- (5) Use library routines - ESSL, PESSL, MASS
- (6) Attention to details – Tuning Guide (SC09-1705-01)
- (7) SMP : pthreads API and parallel directives

What is Wrong with this Code?

```
#include <stdio.h>

int main(void)
{
    double x = 2.0;
    printf("sqrt(x) = %lf\n", sqrt(x));
    return(0);
}
```

This code produces the wrong answer.

```
#include <stdio.h>
#include <malloc.h>

int main(void)
{
    int * pointer;
    pointer = (int *) malloc(sizeof(int));
    *pointer = 3;
    printf("*pointer = %d\n", *pointer);
    return(0);
}
```

This one core-dumps in 64-bit mode (-q64).

Suggestion for C : use lint .

Time Sampled Profiling - Xprofiler

1. Compile and link with “-g -pg” flags and optimization.
2. Run the code, a gmon.out file is produced at the end.
3. Analyze the gmon.out file with gprof or xprofiler.

syntax: gprof > gprof.report
xprofiler your.exe gmon.out

4. Important factors

On AIX, the time-sampling interval = 0.01 sec.

Must sample a realistic problem.

PE/MPI codes generate N gmon.out files.

PVM codes require a special hostfile (specify wd).

A notation is made for every function call = overhead.

Sample CM Hostfile for Profiling

```
#host configuration for v08 nodes
#
#executable path for all hosts
*
ep=$PVM_ROOT/bin/$PVM_ARCH:$PVM_ROOT/lib/$PVM_ARCH:
$CM_AHOME/bin:$CM_LAHOME/es:$PVM_ROOT/lib
v08l01 wd=/cfs/profile/01
v08l02 wd=/cfs/profile/02
v08l03 wd=/cfs/profile/03
v08l04 wd=/cfs/profile/04
v08l05 wd=/cfs/profile/05
v08l06 wd=/cfs/profile/06
v08l07 wd=/cfs/profile/07
v08l08 wd=/cfs/profile/08
v08l09 wd=/cfs/profile/09
v08l10 wd=/cfs/profile/10
v08l11 wd=/cfs/profile/11
v08l12 wd=/cfs/profile/12
v08l13 wd=/cfs/profile/13
v08l14 wd=/cfs/profile/14
v08l15 wd=/cfs/profile/15
v08l16 wd=/cfs/profile/16
```

Alternative : add a chdir() statement to the application code.

Xprofiler Tips

Simplest when gmon.out, binary, and source are in one dir.

Libraries must match across systems.

Select "Report" for the flat profile, click on a routine.

Select "Code Display" for a source-code view.

Use "File ; Set File Search Paths" to set source directory.

Graphical Display:

width of a bar \propto time including called routines
height of a bar \propto time excluding called routines

select : Filter ; Uncluster Functions
then Filter ; Hide All Library Calls

select : Filter ; Filter by CPU Time, or Filter by Call Count

If xprofiler fails with “bad font” error message:

- (1) edit /usr/lib/X11/app-defaults/Xprofiler
replace *narc*font: -ibm-block-...
with *narc*font: fixed
- (2) xrdb –load /usr/lib/X11/app-defaults/Xprofiler

Power-3 Processor (630)

64-bit PowerPC architecture

3 fixed-point units : two for single-cycle ops (add, and, or, ...)
 one for multi-cycle ops (div, ...)

2 floating-point units

2 load/store units

branch history table with dynamic branch prediction

4 instructions dispatched/completed per cycle

out-of-order execution, in-order completion

L1 Data cache : 64 Kbytes, 128-way set associative

L1 Instruction cache: 32 Kbytes

L2 cache 8 Mbytes 4-way set associative

Application performance @ 375 MHz

SPECfp95 51

SPECint95 24

Power-3 Cycle Times

Latency = number of cycles from when an instruction starts executing to when the result is available to dependent instructions

operation	hardware unit	latency (cycles)
add, and, ...	xsu	1
or, nor	xsu	1
b, bc	ifu	1
divw	xmu	21 (not pipelined)
fadd, fmadd	fpu	3-4
fctiwz, frsp	fpu	3-4
fmul, fmuls	fpu	3-4
fdiv	fpu	18-25 (not pipelined)
fdivs	fpu	14-21 (not pipelined)
fsqrt	fpu	22-31 (not pipelined)
fsqrts	fpu	14-23 (not pipelined)
ld, lfd	ldst	2
std, stfd	ldst	1

floating-point units:

double-precision fma: 3-cycle latency, 1-cycle throughput
single-precision fma: 3-cycle latency, 1-cycle throughput

32 64-bit registers

24 64-bit rename registers

Floating-point performance : dgemm 750 Mflops/cpu
(at 200 MHz) op count = $2 * n^3$

Hints from Hardware

If data is not in cache, a load fills a cache line, and there is a significant time-delay. Stride-1 access is best. Reuse data in registers as much as possible. Minimize stores.

Instructions are pipelined - pipeline length - 3 to 4 cycles for floating-point ops on power-3. Each floating point unit can deliver one result per cycle, but data dependency can reduce this to one result in 3 to 4 cycles. Best to schedule enough independent work to keep the throughput close to one op completed per cycle.

Avoid division and square-roots when possible. Use compiler flags to ensure that hardware square roots are used when available.

Help from the Compiler

1. Optimization levels

default	= none
-O	= standard optimizations
-O2	= same as -O
-O3 -qstrict	= more opt., preserves semantics
-O3	= aggressive (test it!)
-O4	= -O3 plus auto arch/tune plus ipa

2. Architecture-specific instructions

default	= common RS6K architecture
-qarch=	-qtune= special instructions (partial list)
pwr2	pwr2s fsqrt, fcirz
pwr2	pwr2 fsqrt, fcirz, quad load/store
ppc	604 sp arithmetic, fctiwz
pwr3	pwr3 sp arithmetic, fctiwz, fsqrt, fsel

3. C aliasing: -qansialias (default for xlc, not cc)

4. F90 code : -qhot -qalias=noaryovrlp

5. Function inlining: -Q, -Q+func, -qipa=level=2

6. Information: -qlist -qsource (read the listing!)

Assembler Language Reference SC23-2642-01

7. Single-precision floating point (power-2, p2sc)

-qfloat=hssngl
-qfloat=hsflt (test it!)

Guidelines for Inlining Functions

Profile the application.

Inline only very short functions that are executed very many times (time per call < 1 microsec).

Within the same source file use:

-Q The compiler will decide (not ideal).

-Q+function1:function2 You specify functions (better).

Across different source files, must use -qipa .

-qipa can dramatically increase compile and link times, and often slows down the code. Consult the compiler manual. Try -qipa=level=2.

My technique: combine source files, use -Q+func1:func2

Libraries

- (1) ESSL : BLAS, linear equation solvers, eigensystems, FFTs, interpolation, integration, random numbers, ...

SMP enabled BLAS, multidimensional FFTs, ...
just link with `-lesslsmp`

www.rs6000.ibm.com/resource/aix_resource/sp_books/

- (2) PESSL : BLAS, linear equation solvers, eigensystems, FFTs, random numbers, ...

SMP enabled PBLAS, multidimensional FFTs, ...
just link with `-lesslsmp -lpesslsmp -lblacssmp`

www.rs6000.ibm.com/resource/aix_resource/sp_books/

- (3) MASS : fast scalar and vector functions (exp, log, ...)

current version 2.6 includes thread-safe vector functions
and thread-safe scalar routines

www.rs6000.ibm.com/resource/technology/MASS/index.html

Example : Matrix Multiplication

Simple Loops: MFlops = 14 for n=480 (silver node)

```
for (i=0; i<n; i++) {  
    for (j=0; j<n; j++) {  
        tmp = 0.0;  
        for (k=0; k<n; k++) {  
            tmp = tmp + A[i*n+k] *B[k*n+j];  
        }  
        C[i*n+j] = tmp;  
    }  
}
```

ESSL : MFlops = 360 (std), 1360 (4-way smp)

```
dgemm( "T" , "T" , n , n , n , 1.0 , A , n , B , n , 0.0 , C , n ) ;
```

Blocking, 4x4 un-rolling : MFlops = 1240 (4-way smp)

* * * * | * * * * | * * * *
* * * * | * * * * | * * * *
* * * * | * * * * | * * * *
* * * * | * * * * | * * * *
-----|-----|-----
* * * * | * * * * | * * * *
* * * * | * * * * | * * * *
* * * * | * * * * | * * * *
* * * * | * * * * | * * * *
-----|-----|-----
* * * * | * * * * | * * * *
* * * * | * * * * | * * * *
* * * * | * * * * | * * * *
* * * * | * * * * | * * * *

A

*

* * * * | * * * * | * * * *
* * * * | * * * * | * * * *
* * * * | * * * * | * * * *
* * * * | * * * * | * * * *
-----|-----|-----
* * * * | * * * * | * * * *
* * * * | * * * * | * * * *
* * * * | * * * * | * * * *
* * * * | * * * * | * * * *
-----|-----|-----
* * * * | * * * * | * * * *
* * * * | * * * * | * * * *
* * * * | * * * * | * * * *
* * * * | * * * * | * * * *

B

blocksize**2 * sizeof(element) < cache size

Information on Cache Blocking

IBM Optimization and Tuning Guide (SC09-1705-01)

IBM Power-3 Redbook (SG24-5155-00)

www.redbooks.ibm.com

Automatic blocking : www.netlib.org/atlas/index.html

Static blocking : www.netlib.org/blas/gemm_based
[ftp.enseeiht.fr/pub/numerique/BLAS/RISC](ftp://ftp.enseeiht.fr/pub/numerique/BLAS/RISC)

Blocking matters most for large matrix problems with stride N access, and where N is > the number of cache lines. Can use a cache-blocked transpose to convert to stride 1 and/or tuned library routines.

MASS Example - Flat Profile

%time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
24.6	683.53	683.53	1747846	0.39	0.39	.fcttrmsa [7]
14.5	1085.28	401.75				._atan2 [8]
10.2	1368.01	282.73				.scalb [9]
6.9	1558.99	190.98				._log [10]
6.7	1744.47	185.48				._exp [11]
5.2	1887.72	143.25				._sin [12]
4.8	2019.94	132.22				._cos [13]
4.4	2142.43	122.49				.kickpipes [14]
3.9	2250.59	108.16				.logb [15]
2.5	2319.94	69.35				.copysign [16]
2.1	2378.50	58.56				.__itrunc [17]
2.1	2435.75	57.25				._xlzexpv [18]
1.5	2476.44	40.69				.tbmx_newpkts
0.9	2500.93	24.49				.free_y [20]
0.9	2524.81	23.88				.zcopy [21]
0.8	2547.57	22.76	2522	9.02	9.02	.initrows [22]
0.8	2569.86	22.29	3075	7.25	7.25	.initnrtrm [23]
0.7	2589.24	19.38				.mpci_recv [24]
0.6	2606.70	17.46				._pxldatn2 [25]
0.6	2623.36	16.66				.malloc_y [26]
0.6	2639.33	15.97	2514	6.35	6.35	.inittst [27]
0.6	2654.81	15.48				.isnan [28]
0.5	2668.36	13.55	77954	0.17	0.17	.initcols [29]
0.3	2677.47	9.11	2514	3.62	286.98	.fillzcol [6]
0.3	2685.35	7.88				.\$SAVEF31 [31]
0.2	2691.93	6.58	2514	2.62	308.54	.fillzblk [5]
0.2	2698.42	6.49	873923	0.01	0.01	.cpltrms [35]

Lots of atan2, log, exp, sin, cos - use MASS

Some Tricks

Loop un-rolling to break a data dependency.

original loop:

```
do i = 1, n
    f = x(i) - dble(int(x(i)+0.5d0))
    sum = sum + f
end do
```

A fast (but dangerous) nearest-integer function.
(-qstrict is required)

```
real*8 rnd, f, sum, x(n)
rnd = 2.0d0**52 + 2.0d0**51
do i = 1, n
    f = x(i) - (rnd + x(i) - rnd)
    sum = sum + f
end do
```

Horner's rule

```
a*x**5 + b*x**4 + c*x**3 + d*x**2 + e*x + f
( (( (a*x+b)*x+c)*x+d)*x+e)*x+f
```

Many more examples:

Optimization and Tuning Guide (SC09-1705-01)

POWER-3 Redbook (SG24-5155)
www.redbooks.ibm.com

Example – a "simple" loop

```
float scale;
float *x, *y;
int i, i1, i2, m;

for (i=i1; i<i2; i++)
{
    y[i] += scale*(-x[i-m] + 2.0*x[i] - x[i+m])
}
```

Time (sec)	Flags	code	lang
1.775	None	original	C
0.306	-O	original	C
0.242	-O -qarch=pwr3	modified	C
0.211	-O -qarch=pwr3 -qunroll=4	modified	C
0.144	-O3 -qarch=pwr3 -qunroll=4, disjoint	modified	C
0.154	-O3 -qarch=pwr3	original	F/C
0.080	-O3 -qarch=pwr3	threaded	F/C

To generate an assembler listing: -qsyntax -qlist

Assembler Language Reference SC23-2642-01

For C try : #pragma disjoint (*x, *y)

Listing – Source Code Section

```
IBM C and C++ Compilers Version 3.6.6.0 --- loop.c 04/14/99 13:40:50

>>>> SOURCE SECTION <<<<
1 | void loop(int *pi1, int *pi2, int *pm, float *pscale, float *x,
           float *y)
2 | {
3 |     int i, i1, i2, m;
4 |     float scale;
5 |
6 |     i1 = *pi1;
7 |     i2 = *pi2;
8 |     m = *pm;
9 |     scale = *pscale;
10 |
11 |    for (i=i1; i<i2; i++)
12 |    {
13 |        y[i] = y[i] + scale*(-x[i-m] + 2.0*x[i] - x[i+m]);
14 |    }
15 |
16 |    return;
17 | }
```

>>>> OPTIONS SECTION <<<<

Get line number information from the source-code section.

In this example, focus on line # 13.

Listing – the Object Section (1)

Original source, compiler flags : none

line#	instruction				
11	000064 bc	4080006C	1	BF	CL.2,cr0,0x1/lt ,
11				CL.1:	
13	000068 lwz	8061008C	1	L4A	gr3=y(gr1,140)
13	00006C lwz	80C10040	0	L4A	gr6=i(gr1,64)
13	000070 rlwinm	54C4103A	2	SLL4	gr4=gr6,2
13	000074 lfsx	7C23242E	1	LFS	fp1=(*)float(gr3,gr4,0)
13	000078 lfs	C0410050	1	LFS	fp2=scale(gr1,80)
13	00007C lwz	80A10088	0	L4A	gr5=x(gr1,136)
13	000080 lwz	80E1004C	1	L4A	gr7=m(gr1,76)
13	000084 subfc	7D073010	2	S	gr8=gr6,gr7
13	000088 rlwinm	5508103A	1	SLL4	gr8=gr8,2
13	00008C lfsx	7C65442E	1	LFS	fp3=(*)float(gr5,gr8,0)
13	000090 lfsx	7C85242E	1	LFS	fp4=(*)float(gr5,gr4,0)
13	000094 lfs	C0BF0000	0	LFS	fp5+=CONSTANT_AREA(gr31,0)
13	000098 fmsub	FC641978	1	FMS	fp3=fp3-fp5,fcr
13	00009C add	7CC63A14	0	A	gr6=gr6,gr7
13	0000A0 rlwinm	54C6103A	1	SLL4	gr6=gr6,2
13	0000A4 lfsx	7C85342E	1	LFS	fp4=(*)float(gr5,gr6,0)
13	0000A8 fsub	FC632028	1	SFL	fp3=fp3,fp4,fcr
13	0000AC fmadd	FC2208FA	2	FMA	fp1=fp1-fp3,fcr
13	0000B0 frsp	FC200818	2	CVLS	fp1=fp1,fcr
13	0000B4 stfsx	7C23252E	0	STFS	(*)float(gr3,gr4,0)=fp1
11	0000B8 lwz	80610040	0	L4A	gr3=i(gr1,64)
11	0000BC addi	38630001	0	AI	gr3=gr3,1
11	0000C0 stw	90610040	0	ST4A	i(gr1,64)=gr3
11	0000C4 lwz	80810048	0	L4A	gr4=i2(gr1,72)
11	0000C8 cmp	7C032000	1	C4	cr0=gr3,gr4
11	0000CC bc	4180FF9C	1	BT	CL.1,cr0,0x1/lt ,
11				CL.3:	
11				CL.2:	
17				CL.4:	
17	0000D0 lwz	83E1005C	1	L4A	gr31=#stack(gr1,92)

Contains integer loads and explicit address computation. There is too much work per loop iteration, and there is no unrolling.

Elapsed-time = 1.775 sec.

Listing – the Object Section (2)

Original source, compiler flags : -O

0	000064	bc	4240002C	0	BCF	ctr=CL.17,
11					CL.1:	
13	000068	lfsu	C4840004	1	LFSU	fp4, gr4=(*) float(gr4, 4)
13	00006C	fmsub	FC641838	1	FMS	fp3=fp3, fp4, fp0, fcr
13	000070	lfsu	C4A70004	0	LFSU	fp5, gr7=(*) float(gr7, 4)
13	000074	fsub	FC632828	3	SFL	fp3=fp3, fp5, fcr
13	000078	fmadd	FC6110FA	2	FMA	fp3=fp2, fp1, fp3, fcr
13	00007C	lfs	C0460008	0	LFS	fp2=(*) float(gr6, 8)
13	000080	frsp	FC601818	2	CVLS	fp3=fp3, fcr
13	000084	stfsu	D4660004	0	STFSU	gr6, (*) float(gr6, 4)=fp3
13	000088	lfsu	C4630004	0	LFSU	fp3, gr3=(*) float(gr3, 4)
0	00008C	bc	4200FFDC	0	BCT	ctr=CL.1,
0					CL.17:	
13	000090	lfsu	C4840004	1	LFSU	fp4, gr4=(*) float(gr4, 4)

Much cleaner code, but there is a frsp, and no unrolling.

Elapsed time = 0.306 sec.

Listing – the Object Section (3)

Modified code, compiler flags: -O -qarch=pwr3 -qunroll=4

0	0000D4 bc	42400088	0	BCF	ctr=CL.27,
11				CL.1:	
13	0000D8 lfs	C0A70004	1	LFS	fp5=(*) float(gr7,4)
13	0000DC lfs	C0C50004	1	LFS	fp6=(*) float(gr5,4)
13	0000E0 fmsubs	ECA53878	1	FMSS	fp5=fp7, fp5, fp1, fcr
13	0000E4 fsubs	ECA53028	4	SFS	fp5=fp5, fp6, fcr
13	0000E8 fmadds	ECA0117A	4	FMAS	fp5=fp2, fp0, fp5, fcr
13	0000EC lfs	C0480014	0	LFS	fp2=(*) float(gr8,20)
13	0000F0 stfs	D0A80004	0	STFS	(*) float(gr8,4)=fp5
13	0000F4 lfs	C0A30008	0	LFS	fp5=(*) float(gr3,8)
13	0000F8 lfs	C0C70008	0	LFS	fp6=(*) float(gr7,8)
13	0000FC lfs	C0E50008	0	LFS	fp7=(*) float(gr5,8)
13	000100 fmsubs	ECA62878	1	FMSS	fp5=fp5, fp6, fp1, fcr
13	000104 fsubs	ECA53828	4	SFS	fp5=fp5, fp7, fcr
13	000108 fmadds	ECA0197A	4	FMAS	fp5=fp3, fp0, fp5, fcr
13	00010C lfs	C0680018	0	LFS	fp3=(*) float(gr8,24)
13	000110 stfs	D0A80008	0	STFS	(*) float(gr8,8)=fp5
13	000114 lfs	C0A3000C	0	LFS	fp5=(*) float(gr3,12)
13	000118 lfs	C0C7000C	0	LFS	fp6=(*) float(gr7,12)
13	00011C lfs	C0E5000C	0	LFS	fp7=(*) float(gr5,12)
13	000120 fmsubs	ECA62878	1	FMSS	fp5=fp5, fp6, fp1, fcr
13	000124 fsubs	ECA53828	4	SFS	fp5=fp5, fp7, fcr
13	000128 fmadds	ECA0417A	4	FMAS	fp5=fp8, fp0, fp5, fcr
13	00012C lfs	C108001C	0	LFS	fp8=(*) float(gr8,28)
13	000130 stfs	D0A8000C	0	STFS	(*) float(gr8,12)=fp5
13	000134 lfsu	C4A30010	0	LFSU	fp5, gr3=(*) float(gr3,16)
13	000138 lfsu	C4E70010	0	LFSU	fp7, gr7=(*) float(gr7,16)
13	00013C lfsu	C4C50010	0	LFSU	fp6, gr5=(*) float(gr5,16)
13	000140 fmsubs	ECA72878	1	FMSS	fp5=fp5, fp7, fp1, fcr
13	000144 fsubs	ECA53028	4	SFS	fp5=fp5, fp6, fcr
13	000148 fmadds	ECA0217A	4	FMAS	fp5=fp4, fp0, fp5, fcr
13	00014C lfs	C0880020	0	LFS	fp4=(*) float(gr8,32)
13	000150 stfsu	D4A80010	0	STFSU	gr8, (*) float(gr8,16)=fp5
13	000154 lfs	C0E30004	0	LFS	fp7=(*) float(gr3,4)
0	000158 bc	4200FF80	0	BCT	ctr=CL.1,
0				CL.27:	
13	00015C lfs	C0C50004	1	LFS	fp6=(*) float(gr5,4)

unrolled, but instructions are not optimally scheduled

Elapsed time = 0.211 sec.

Listing – the Object Section (4)

Fortran source, compiler options : -O3 -qarch=pwr3

0	000160	bc	419E0088	0	BT	CL.25,cr7,0x4/eq ,
5					CL.0:	
6	000164	fmaadds	ED85207A	1	FMAS	fp12=fp4,fp5,fp1,fcr
6	000168	lfs	C0850004	1	LFS	fp4=x(gr5,4)
6	00016C	fmsubs	ECA74038	1	FMSS	fp5=fp8,fp7,fp0,fcr
6	000170	lfs	C3A30010	1	LFS	fp29=y(gr3,16)
6	000174	fmsubs	ED695038	1	FMSS	fp11=fp10,fp9,fp0,fcr
6	000178	lfsu	C5A50008	1	LFSU	fp13,gr5=x(gr5,8)
6	00017C	lfs	C0E40004	1	LFS	fp7=x(gr4,4)
6	000180	lfs	C1070004	1	LFS	fp8=x(gr7,4)
6	000184	lfs	C3C50004	1	LFS	fp30=x(gr5,4)
6	000188	fsubs	EC822028	1	SFS	fp4=fp2,fp4,fcr
6	00018C	lfsu	C5270008	1	LFSU	fp9,gr7=x(gr7,8)
6	000190	lfsu	C5440008	1	LFSU	fp10,gr4=x(gr4,8)
6	000194	fsubs	EDA36828	1	SFS	fp13=fp3,fp13,fcr
6	000198	lfsu	C7E50008	1	LFSU	fp31,gr5=x(gr5,8)
6	00019C	lfs	C063000C	1	LFS	fp3=y(gr3,12)
6	0001A0	fsubs	EFC5F028	1	SFS	fp30=fp5,fp30,fcr
6	0001A4	stfs	D0C30004	1	STFS	y(gr3,4)=fp6
6	0001A8	fmsubs	EC483838	1	FMSS	fp2=fp7,fp8,fp0,fcr
6	0001AC	stfsu	D5830008	1	STFSU	gr3,y(gr3,8)=fp12
6	0001B0	lfs	C0E70004	1	LFS	fp7=x(gr7,4)
6	0001B4	lfs	C1040004	1	LFS	fp8=x(gr4,4)
6	0001B8	fsubs	ECABF828	1	SFS	fp5=fp11,fp31,fcr
6	0001BC	lfs	C183000C	1	LFS	fp12=y(gr3,12)
6	0001C0	fmaadds	ECCDE87A	1	FMAS	fp6=fp29,fp13,fp1,fcr
6	0001C4	fmaadds	EDA4187A	1	FMAS	fp13=fp3,fp4,fp1,fcr
6	0001C8	lfs	C0830010	1	LFS	fp4=y(gr3,16)
6	0001CC	fmsubs	EC695038	1	FMSS	fp3=fp10,fp9,fp0,fcr
6	0001D0	lfsu	C5440008	1	LFSU	fp10,gr4=x(gr4,8)
6	0001D4	lfsu	C5270008	1	LFSU	fp9,gr7=x(gr7,8)
6	0001D8	stfs	D1A30004	1	STFS	y(gr3,4)=fp13
6	0001DC	stfsu	D4C30008	1	STFSU	gr3,y(gr3,8)=fp6
6	0001E0	fmaadds	ECDE607A	1	FMAS	fp6=fp12,fp30,fp1,fcr
0	0001E4	bc	4200FF80	0	BCT	ctr=CL.0,
0					CL.25:	
6	0001E8	fmr	FD602090	1	LRFL	fp11=fp4

good instruction mix

Elapsed time = 0.154 sec.

OpenMP Directives

www.openmp.org OpenMP specification

OpenMP supported in: Fortran xlf version 6 and up
C for AIX version 5.0 and up
C++ - try Kuck Associates

Info on IBM xlf implementation: ACTC web page

<http://www.research.ibm.com/actc/Talks/RaulSilvera/index.htm>

Easy work-sharing directives (but hard to scope variables):

Fortran : !\$omp parallel
 !\$omp do
 !\$omp end parallel

 !\$omp parallel do

C: #pragma omp parallel
 {
 #pragma omp for
 }

 #pragma omp parallel for

Compile and link with xlf_r or cc_r and -qsmp

-qsmp=noauto (disables automatic parallelization)
-qreport=smplist (Fortran only)
-qnosave (may help for xlf_r)

OpenMP Directives: C Example

```
/*-----*/
/* use simple loops to do matrix multiplication */
/*-----*/
#pragma omp parallel
{
#pragma omp for
    for (i=0; i<n; i++)
    {
        for (j=0; j<n; j++) Bt[j*n+i] = B[i*n+j];
    }
#pragma omp for
    for (i=0; i<n; i++)
    {
        for (j=0; j<n; j++)
        {
            tmp = 0.0;
            for (k=0; k<n; k++)
            {
                tmp = tmp + A[i*n+k] * Bt[j*n+k];
            }
            C[i*n+j] = tmp;
        }
    }
} /* end parallel region */
```

Compile and link with cc_r -qsmp=noauto

OpenMP Directives : Fortran Example

```
!$OMP PARALLEL DO
!$OMP& default(private)
!$OMP& shared(x,y,z,nblist,nbcount,dcutsq)
do j = 1, n
    count = 0
    do i = 1, n
        dx = x(i) - x(j)
        dy = y(i) - y(j)
        dz = z(i) - z(j)
        dsq = dx*dx + dy*dy + dz*dz
        if (dsq.lt.dcutsq .and. i.ne.j) then
            count = count + 1
            nblist(count,j) = i
        endif
    end do
    nbcount(j) = count
end do
```

Compile and link with xlf_r -qsmp=noauto

Note: OpenMP implicitly has default(shared).

Must be careful about variable scope (shared vs private).

OpenMP SPMD Example

```
program spmd
implicit none

!$OMP PARALLEL
    call setup_threads()
    call do_work()
!$OMP END PARALLEL
end

subroutine setup_threads()
implicit none
include 'thread_common.h'
integer omp_get_thread_num
integer omp_get_num_threads
tid = omp_get_thread_num()
nth = omp_get_num_threads()
end

subroutine do_work()
implicit none
include 'thread_common.h'
write(6,*) 'tid, nth = ', tid, nth
end
```

thread_common.h :

```
integer tid, nth
common /thread_common/ tid, nth
!$OMP THREADPRIVATE (/thread_common/)
```

OpenMP Tips

Use coarse-grain parallel strategy as much as possible.
Avoid overhead to create (or wake up) threads (~100 usec).

Profile the code (xprofiler, gprof, prof, tprof).

Use -qsmp=noauto, and do not specify -qsmp for routines that contain no OpenMP directives.

If the work-load depends on a loop counter, try a guided schedule.

Avoid unnecessary barriers (!\$omp end do nowait).

To control the number of threads:

```
export XLSMPOPTS=parthds=4 (for 4 threads)
```

AIX environment variables that can help:

```
export SPINLOOPTIME=1000 (for mutex contention)  
export MALLOCMULTIHEAP=true
```

SMP runtime variables that can help:

```
export XLSMPOPTS="yields=0:spins=0"
```

Try tools from Kuck Associates (www.kai.com).

OpenMP + MPI

Not well suited for every application.

Limited performance improvement is possible in some cases, relative to pure MPI.

Fortran: mpxlf_r -qsmp=noauto ...

C: mpcc_r -qsmp=noauto ...

Limit the number of threads:

export XLSMPOPTS=parthds=2 (for two threads)

Avoid communication from inside a parallel region.

In some cases, use SMP libraries + MPI (ESSL, PESSL).

Fortran: mpxlf_r ... -lesslsmp

C: mpcc_r ... -lesslsmp -lpesslsmp -lblacssmp

IBM Redbook "Scientific Applications on RS/6000 SP Environments" SG24-5611-00

Default Pthread Attributes – AIX 4.3

`pthread_attr_init(&attr); // initialize with default values`

detach state : detached (not joinable)

contention scope : process (not system)

inherit schedule : inherited (not fixed)

schedule priority : 1 (range 1-127)

schedule policy : other (not round-robin or fifo)

functions to control attributes:

`pthread_attr_setdetachstate`
`pthread_attr_setscope`
`pthread_attr_setinheritsched`
`pthread_attr_setschedparam`
`pthread_attr_setschedpolicy`

Template for Thread Creation

```
#include <pthread.h>
#include <stdlib.h>

void * thread_function(void *);

int numthreads;
int * tid;

pthread_t * thread;
pthread_attr_t attr;

/*-----
/* set-up for joinable threads with system scope */
/*-----*/
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr,
                           PTHREAD_CREATE_JOINABLE);
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

tid = (int *)malloc(numthreads*sizeof(int));

thread = (pthread_t *)malloc(numthreads*sizeof(pthread_t));

/*-----
/* create threads */
/*-----*/
for (i=0; i<numthreads; i++)
{
    tid[i] = i;
    pthread_create(&thread[i], &attr, thread_function,
                  (void *)&tid[i]);
}

/*-----
/* wait for all threads to finish */
/*-----*/
for (i=0; i<numthreads; i++) pthread_join(thread[i], NULL);
```

Template for the Thread Function

```
int global_variable; //shared by all threads
int ng = 100;         //shared by all threads

void * thread_function(void * pArg)
{
    int local_variable; //local to each thread
    int * local_array; //local to each thread
    int mytid;          //local to each thread
    int nl;              //local to each thread

    mytid = * ((int *) pArg);
    nl = ng*mytid;

    local_array = (int *) alloca(nl*sizeof(int));

    do_work();

}
```

Variables declared outside are shared.

Variables declared inside are thread-local.

Use malloc for global shared memory.

Use alloca for thread-local dynamic allocation, or use malloc inside the thread function with locally declared pointers.

must add the compiler flag “-ma” for alloca

alloca is not directly portable

Optimization Tips

- (1) Watch data access patterns. Allocate memory in contiguous blocks, access with stride-1 when possible, and avoid arrays of pointers.
- (2) Profile and/or trace for detailed timing info.
- (3) Look for opportunities to inline functions.
- (4) In performance-critical routines, use as a minimum the compiler flags “-O3 -qstrict -qarch=... -qtune=...”. For C add “-qansialias” when appropriate; for F90 try “-qhot” and "-qalias=noaryovrlp” when appropriate.
- (5) Read the compiler listing : “-qlist -qsource”
- (6) Experiment.